UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Prototipo de infraestructura de ayuda al ciudadano para la notificación de incidencias en el reciclado

Autor/es

SERGIO MAULEÓN CRISTÓBAL

Director/es

JESÚS MARÍA ARANSAY AZOFRA y ELOY JAVIER MATA SOTÉS ,

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2017-18

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Prototipo de infraestructura de ayuda al ciudadano para la notificación de incidencias en el reciclado

Realizado por:

Sergio Mauleón Cristóbal

Tutorizado por:

Eloy Javier Mata Sotés

Jesús María Aransay Azofra

Logroño, febrero, 2018

0.1. Resumen

Este trabajo de fin de grado consiste en la implementación de un sistema de envío de incidencias que permita a los ciudadanos informar acerca de desperfectos en los contenedores de reciclaje de su zona de residencia. Para ello, este trabajo se compondrá de dos proyectos: una aplicación `iOS`, con la que los usuarios interactuarán para enviar las incidencias; y un panel de administración web, a través del cual se gestionarán las incidencias enviadas.

Estos dos proyectos crearán una funcionalidad completa, ya que el usuario podrá enviar incidencias que deban ser resueltas a través de la aplicación móvil, y los administradores, mediante el panel de administración, las pondrán en conocimiento de la autoridad correspondiente para su gestión.

0.2. Abstract

This final degree project consists in the implementation of a system for sending incidences that allow the citizens to inform about damages in the recycling containers nearby their residence area. To create the functionality of the system, it's been needed two projects: an application for iOS devices, with which the users will interact to send the incidences; and a web administration panel, through which the incidences will be managed.

These two projects will create a full functionality, as the user will be able to send incidences that have to be resolved through the mobile app, and the administrators, by means of the administration panel, will inform the corresponding authority for their management.

Índice

0.3. Antecedentes

Este trabajo de fin de grado se ha acordado dentro de un convenio establecido con *TheCircularLab*. Este centro es, según consta en su página web, el único centro de innovación en el mundo especializado en economía circular. En este centro se incide en todas las fases del ciclo de vida de los envases: desde su concepción, a través del ecodiseño, hasta su reintroducción al ciclo de consumo a través de nuevos productos.

En este laboratorio de innovación del envase se estudia, concibe, prueba y aplica en un entorno real las mejores prácticas. Todo ello en un marco de estrecha colaboración de innovación abierta entre empresas, administraciones públicas y ciudadanos.

TheCircularLab es un proyecto pionero de Ecoembes, que es la organización que cuida del medio ambiente a través del reciclaje y el ecodiseño de los envases en España. Hacen posible que los envases de plástico, latas y briks (contenedor amarillo) y los envases de cartón y papel (contenedor azul) puedan tener una segunda vida.

Ecoembes se creó en 1996, adelantándose a la Ley 11/97 de Envases y Residuos de Envases. Esta ley establece unas obligaciones que pretenden la recuperación de los residuos de los envases, su posterior tratamiento y valorización. Desde que se creó, han evitado la emisión de más de 17,7 millones de toneladas de CO_2 a la atmósfera, lo que supone un ahorro de 7 millones de MWh**,** lo mismo que consumirían los automóviles que hay en España durante los próximos 40 años.

En el 2016 se reciclaron en España el 82,3% de envases de cartón y papel, y el 66,5% de envases de plástico, lo que significa que se alcanzó una tasa de reciclaje del 76%.

Ecoembes, gracias a su actividad, genera más de 42.600 puestos de trabajo en España.

En junio de 2016 empecé un periodo de seis meses de prácticas en *TheCircularLab* junto con otros nueve jóvenes. Durante este periodo, se nos lanzó un reto: idear, diseñar y desarrollar una aplicación para dispositivos Android que incentivara al ciudadano a reciclar.

Durante el proceso de ideación del contenido, se propusieron las funcionales para la aplicación. Al no poder desarrollarse todas las ideas que se sugirieron, se me ofreció la posibilidad de crear un producto mínimo viable para dispositivos `iOS`, de objetivo similar al de su equivalente en `Android`, pero implementando una funcionalidad distinta.

0.4. Objetivo

El problema de origen al iniciar este proyecto es: ¿qué medidas se pueden tomar para incentivar a los ciudadanos a reciclar? Uno de los problemas con los que se encuentra parte de la población es que no tienen contenedores específicos para el reciclaje de los distintos residuos cerca de su domicilio, o que los que hay están en mal estado.

El objetivo de este proyecto es crear una aplicación para dispositivos `iOS` que solucione el problema anterior. Para ello, la aplicación permitirá al usuario enviar información sobre los impedimentos relacionados con los contenedores de reciclaje que se encuentran en su entorno.

A su vez, se desarrollará un panel de administración web que permita gestionar las incidencias enviadas por los usuarios, y reportarlas a los organismos encargados de solucionarlas.

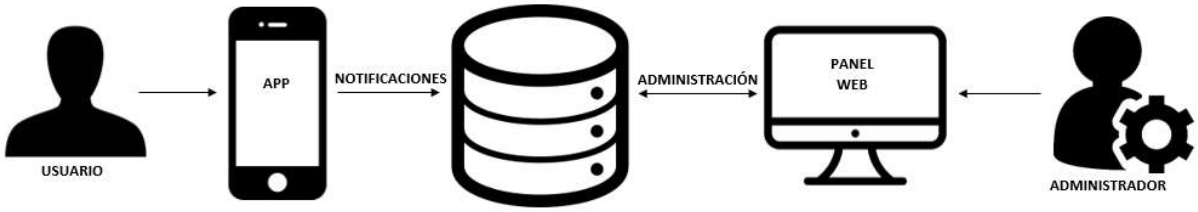

Figura 1: Interacción entre los proyectos

En la figura 1 se muestran las relaciones que se establecerán entre los distintos componentes del proyecto. Como objeto principal se sitúa la base de datos, que será el nexo entre la aplicación `iOS` y el panel de administración web. Los usuarios, a través de la aplicación, enviarán información a la base de datos, y los administradores, a través del panel web, accederán a esos datos enviados por los usuarios para gestionarlos.

0.5. Participantes

A continuación, se introducirán a las personas involucradas en este proyecto y la labor que desempeñarán en el mismo.

- **Sergio Mauleón Cristóbal**: Alumno de Grado de Ingeniería Informática y desarrollador del TFG.
- **Jesús María Aransay Azofra**: Tutor del TFG. Profesor del Departamento de Matemáticas y Computación.
- **Eloy Javier Mata Sotés**: Tutor del TFG. Profesor del Departamento de Matemáticas y Computación.
- **Zacarías Torbado Martínez**: Coordinador de *TheCircularLAB*, representa al cliente.

1. Subproyecto 1: Panel de administración web

1.1. Análisis

Esta parte del proyecto consistirá en el desarrollo de una aplicación web, que actuará como panel de administración que permitirá a determinados usuarios (designados como administradores) acceder y gestionar las incidencias enviadas por los usuarios de la aplicación móvil.

Para este proyecto se van a distinguir dos tipos de administradores, en función del organismo al que representen. El administrador global será el representante del organismo Consorcio de Aguas y Residuos de La Rioja, y podrá añadir nuevas entidades de las clases definidas en la base de datos, así como acceder al listado completo de incidencias, organismos, municipios y administradores.

El otro tipo de administradores serán los específicos para cada municipio. Sólo podrán acceder a las incidencias relativas a su zona, ver el detalle del organismo al que están asociados y poder editarlo, y ver sus incidencias representadas en la vista del mapa.

El proceso de captura de requisitos se ha llevado a cabo a través de reuniones con el cliente (véase anexo 1).

1.1.1. Análisis de los requisitos funcionales

1. El sistema será capaz de almacenar las incidencias enviadas por los usuarios.
2. Para acceder al sistema de administración, los usuarios deberán identificarse a través de un login, introduciendo un nombre de usuario y contraseña.
3. El sistema determinará la zona geográfica de cada incidencia automáticamente en función de sus coordenadas GPS, y será asignará al organismo responsable de su gestión.
4. El sistema diferenciará distintos administradores, correspondientes a las distintas zonas geográficas en las que se pueden clasificar las incidencias.
5. En el sistema habrá un administrador global, capaz de acceder a todas las incidencias.
6. El sistema, a través de las coordenadas de las incidencias, permitirá a cada administrador ver un mapa con las incidencias de su zona geográfica.
7. Los administradores, a través del sistema, podrán gestionar (aceptar o rechazar) las incidencias correspondientes a su zona. También podrán añadir comentarios a las incidencias, pero no podrán editarlas.

1.1.2. Análisis de los requisitos no funcionales

1. Para realizar el panel de administración, se utilizará el framework `Symfony`, ya que esto facilitará la integración de este proyecto con el resto de proyectos del cliente.
2. El panel de administración deberá visualizarse correctamente en las últimas versiones de los siguientes navegadores:
    - Google Chrome
    - Mozilla Firefox
    - Microsoft Edge
    - Safari

3. Cuando se realicen paradas para realizar funciones de mantenimiento, se ha de garantizar que no se perderá, ni modificará, ningún dato.

1.1.3. Tecnología a utilizar

Como se ha indicado en el requisito no funcional 1 del panel de administración, se desarrollará con el framework Symfony por imposición del cliente. Consiste en un framework **Modelo-Vista-Controlador** de código abierto ideado para desarrollar aplicaciones y servicios web con PHP.

El IDE elegido para gestionar el proyecto Symfony será Eclipse, en su versión para desarrolladores PHP.

Como sistema gestor de base de datos, trabajaremos con MySQL, y como cliente de explotación para la base de datos utilizaremos phpMyAdmin.

Para gestionar las dependencias del proyecto Symfony, se usará Composer.

Como *stack* de soluciones se usará XAMP, que nos permitirá contar tanto con un servidor Apache como con un servidor MySQL.

Se creará un repositorio para mantener el control de versiones del proyecto. He elegido BitBucket ya que tiene un plan gratuito para repositorios ilimitados y equipos de cinco personas o menos. De este modo, podré dar acceso a los repositorios a mis tutores del proyecto. Otra razón por la cual he elegido BitBucket es porque ya he trabajado con esta plataforma durante mi periodo de formación.

1.1.4. Alcance del proyecto

El alcance de este proyecto no contempla los siguientes aspectos:

1. **Mantenimiento**

El mantenimiento del panel de administración una vez entregado al cliente será su responsabilidad, así como las futuras modificaciones de este.

2. **Integración**

El panel de administración no comprende todas las funcionalidades necesarias para la correcta gestión de una aplicación completa desde el punto de vista del cliente. De modo que la posterior migración de la base de datos al servidor del cliente, y la integración de este panel de administración en su sistema de gestión, serán tareas llevadas a cabos por el cliente.

3. **Manual de usuario**

No se entregará un manual de usuario explicando cómo es la navegabilidad o qué funciones cumplen cada uno de los apartados del panel de administración, ya que la información visual que se puede obtener de la propia interfaz es suficiente para su correcto manejo.

1.1.5. Planificación

Para explicar la planificación de este proyecto, se han creado varios documentos en los que se indica cómo se distribuirán las horas de trabajo, cómo se organizará el trabajo a realizar, y cuáles serán las fechas en las que se presentarán las distintas partes que componen el proyecto. Toda esta información está disponible en el Anexo 2. Planificación del proyecto.

1.1.6. Plan de pruebas

A continuación, se indican las pruebas que se realizarán una vez implementada la funcionalidad del proyecto.

- **Inicio de sesión**

Se harán pruebas de inicio de sesión con usuarios que estén almacenados como administradores en la base de datos. El objetivo es comprobar que el sistema discrimina entre usuarios autorizados y no autorizados

- **Edición de una incidencia**

Se accederá al detalle de una incidencia y se cambiará el estado, se añadirá un comentario, y se modificará la imagen. El objetivo es comprobar que estos cambios son realizados correctamente y quedan almacenados en la base de datos.

- **Crear un organismo**

Se creará un organismo a través de la vista destinada a ello, para comprobar que los requisitos para su creación se cumplen y que la base de datos queda actualizada con la nueva entidad.

- **Acceder al detalle de una incidencia a través del mapa**

Esta prueba abordará varias funcionalidades. Se verificará que el mapa se visualice correctamente. También se comprobará que el mapa se muestra centrado sobre el municipio correspondiente al administrador con el que se esté logueado. Por último, se verá si las incidencias del organismo asociado a nuestro usuario se visualizan en al mapa de manera adecuada.

1.2. Diseño

1.2.1. Diseño de la base de datos

Se decidió diseñar e implementar una base de datos con el fin de almacenar la información necesaria para el proyecto por dos motivos. El primero es que habrá información no pensada para ser modificada. Esto se podría haber solucionado haciendo que cada vez que se iniciara la aplicación móvil, se creara esa información de forma local en el dispositivo, pero eso haría aumentar el consumo de recursos del teléfono cada vez que se ejecutara la aplicación.

La segunda y principal razón es que la gestión de las incidencias no se realizará de forma inmediata en el momento en el que los usuarios las envíen. Por ello se necesita un sistema de persistencia que almacene la información. De este modo, los administradores podrán acceder a esa información de forma asíncrona, sin necesidad de estar conectados a la aplicación web en el momento de su envío.

1.2.1.1. Descripción de la base de datos

La base de datos de este proyecto tendrá como entidad principal la tabla **incidencia**. De cada incidencia se querrá saber su imagen, descripción, fecha, latitud, longitud, estado y título. También se guardará el identificador del usuario que ha creado esa incidencia, así como el identificador de la categoría a la que corresponde, y el del organismo encargado de gestionarla.

Los **organismos** representan a los ayuntamientos de los **municipios**, y serán las instituciones encargadas de llevar a cabo la gestión de las incidencias. De cada organismo se guardará su nombre, teléfono, email y la dirección en la que están ubicados. Cada municipio sólo puede estar vinculado a un organismo, y un organismo sólo puede gestionar las incidencias de un municipio

Como este proyecto lo forman dos aplicaciones, se distinguirá entre dos tipos de usuarios, los usuarios del panel web (administradores) y los usuarios de la aplicación `iOS`.

Para los **usuarios de la aplicación** se necesitará saber su nombre y email.

Para los **administradores** del panel web, se almacenará un nombre de usuario, una contraseña y un rol. También contarán con el identificador del organismo del cual son encargados. Cada administrador estará encargado de un único organismo, pero un organismo puede estar gestionado por varios administradores.

Cada incidencia contará con una **categoría**, a través de la cual el usuario podrá clasificar la incidencia. Para cada categoría se querrá saber su nombre y su descripción. Cada incidencia deberá estar asociada a una categoría.

Los **municipios** estarán relacionados únicamente con un organismo, y se almacenará su nombre, latitud y longitud. De este modo, se podrá asignar cada incidencia al organismo correspondiente gracias a la geolocalización.

La **dirección** representará una ubicación, y estará compuesta por una ciudad, una calle y un número.

1.2.1.2. Diagrama de entidad relación

En la figura 2 se muestra el diagrama entidad relación que representa la estructura de la base de datos que se ha descrito en el apartado anterior.

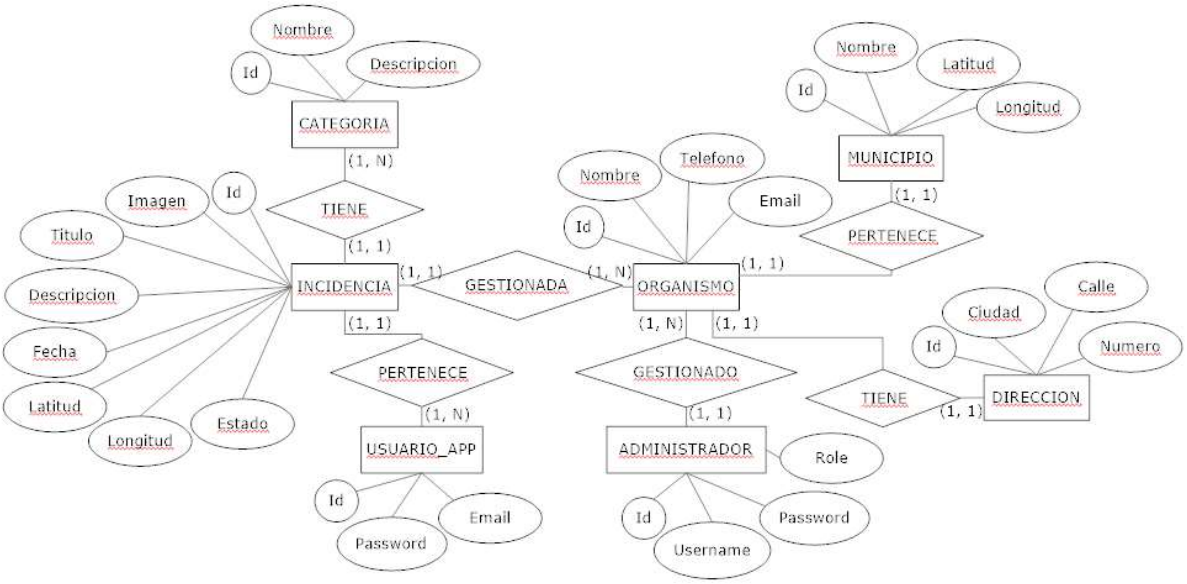

Figura 2: Diagrama entidad relación

Las tablas son las siguientes:

INCIDENCIA

id	imagen	titulo	descripcion	fecha	latitud	...

...	longitud	estado	idCategoria	idOrganismo	idUsuarioApp
			FK: Categoria	FK: Organismo	FK: Usuario_App

Tipos: id, idCategoria, idOrganismo, idUsuarioApp (integer); imagen, titulo, descripcion, estado (varchar); fecha (date); latitud, longitud (double).

ORGANISMO

id	nombre	telefono	email	idDireccion
				FK: Direccion

Tipos: id (integer); nombre, telefono, email (varchar).

CATEGORIA

id	nombre	descripcion

Tipos: id (integer), nombre, descripcion (varchar).

USUARIO_APP

id	nombre	email

Tipos: id (integer), nombre, email (varchar).

ADMINISTRADOR

id	username	password	role	idOrganismo
				FK: Organismo

Tipos: id, idOrganismo (integer); username, password, role (varchar).

MUNICIPIO

id	nombre	latitud	longitud	idOrganismo
				FK: Organismo

Tipos: id, idOrganismo (integer); nombre (varchar); latitud, longitud (double).

DIRECCION

id	ciudad	calle	numero

Tipos: id (integer); ciudad, calle, numero (varchar).

1.2.2. Diseño de la interfaz gráfica

El diseño del panel ha sido impuesto por el cliente, el cual ha proporcionado la plantilla `Metronic`, en la que se establece dónde deben ir los distintos elementos que forman la vista. A continuación, se mostrarán varias de estas interfaces para su explicación.

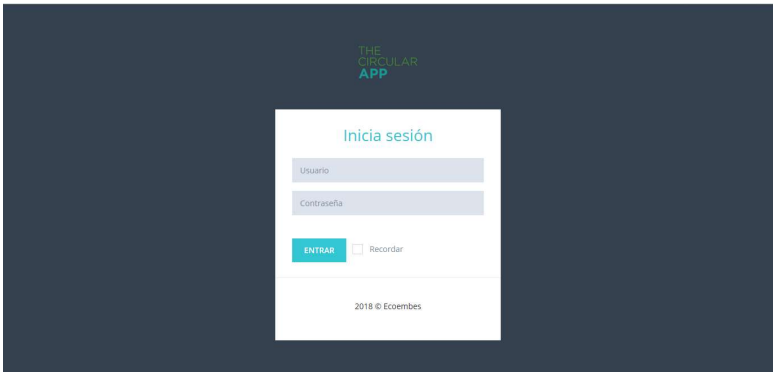

Figura 3: Vista de Login

El logotipo de la vista mostrada en la figura 3 se ha obtenido de la guía de estilos de la empresa, que ha sido desarrollada en el Anexo 4.

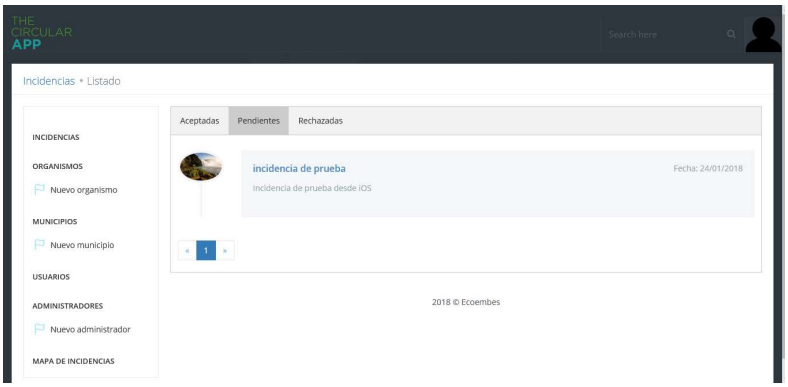

Figura 4: Vista listado de incidencias

La interfaz de la figura 4 muestra el esquema básico del panel de administración. En la parte superior, a la izquierda se sitúa el logotipo del cliente, y a la derecha un buscador y una imagen a través de la cual se puede acceder al perfil del administrador con el que nos hemos logueado, o podremos cerrar la sesión.

En la parte izquierda se sitúa el menú, mostrando las opciones disponibles para el administrador. Estas opciones variarán en función del tipo de administrador que haya iniciado sesión. El administrador del Consorcio de Aguas y Residuos de La Rioja tendrá a su disposición un menú más amplio que los administradores de un municipio concreto, ya que dispone de más funcionalidades. Las diferentes configuraciones del menú se pueden ver en la figura 5.

Figura 5: Diferentes menús para los administradores

En la parte central de la interfaz, se mostrará el contenido correspondiente a cada vista.

1.2.3. Diseño de las clases

El acceso a las entidades de la base de datos del proyecto no se hará directamente a través de peticiones `SQL` contra sus entidades. Para realizar consultas a la base de datos, configuraremos el archivo `parameters.yml`, en el que indicaremos el nombre de la base de datos, host, puerto, usuario y password.

```
# /app/config/parameters.yml
parameters:
       database_host: 127.0.0.1
       database_port: null
       database_name: thecircularapp
       database_user: root
       database_password: root
       mailer_transport: smtp
       mailer_user: null
       mailer_password: null
       secret: ThisTokenIsNotSoSecretChangeIt
       mailer_host: 127.0.0.1
```

Una vez configurado, se podrán crear las clases que mapearán las entidades de la base de datos, gracias al conjunto de librerías `Doctrine`. Estas librerías van a simplificar las operaciones que se realicen contra la base de datos, ya que van a permitir trabajar con un estándar de definición de datos, tablas y relaciones mediante clases `PHP`.

`Doctrine` es un `ORM` (Object Relational Mapper), de modo que puede mapear una base de datos relacional mediante clases. Para crear esas clases se usará la siguiente sentencia en la consola de comandos: `php bin/console doctrine:mapping:import CircularBundle annotation --em=default`, lo que las creará en la carpeta `src/CircularBundle/Entity`. A través de estas clases tendremos las tablas de la base de datos como objetos `PHP`.

A continuación, se muestran las anotaciones de la clase `Incidencia` creadas por `Doctrine` para realizar el mapeo con la tabla homónima de la base de datos.

```
use Doctrine\ORM\Mapping as ORM;
```

Esta sentencia hace que la clase en la que se use esté disponible bajo `@ORM/ClassName`.

```
/**
 * Incidencia
 *
* @ORM\Table(name="incidencia",
indexes={@ORM\Index(name="FK_INCIDENCIA_USUARIO_APP",
columns={"id_usuario_app"}), @ORM\Index(name="FK_INCIDENCIA_CATEGORIA",
columns={"id_categoria"}), @ORM\Index(name="FK_INCIDENCIA_ORGANISMO",
columns={"id_organismo"})})
 * @ORM\Entity
 */
```

En el grupo de anotaciones superior se establece el nombre de la tabla que mapea la clase y las relaciones que establece con el resto de las clases de la base de datos.

```
/**
* @var integer
*
* @ORM\Column(name="id", type="integer", nullable=false)
* @ORM\Id
* @ORM\GeneratedValue(strategy="IDENTITY")
*/
private $id;
```

Para cada campo de la tabla, se crearán atributos en la clase `PHP`. Estos atributos tendrán anotaciones con las que se especifiquen las características del atributo de la tabla al que hacen referencia.

```
/**
* @var \Organismo
*
* @ORM\ManyToOne(targetEntity="Organismo")
* @ORM\JoinColumns({
*   @ORM\JoinColumn(name="id_organismo", referencedColumnName="id")
* })
*/
private $idOrganismo;
```

El campo `$idOrganismo` hace referencia a una relación de la tabla, de modo que se han añadido anotaciones que especifican de qué tipo es la relación (`@ORM\ManyToOne(targetEntity="Organismo")`), y qué columnas de las tablas están involucradas (`@ORM\JoinColumn(name="id_organismo", referencedColumnName="id")`).

Debido a que este proyecto se complementa con una aplicación móvil, las diferentes clases han de poder ser enviadas de forma sencilla a través de peticiones `http`. Para ello, las clases deben implementar la interfaz `JsonSerializable`, y modificar la función `jsonSerializable()` para que devuelva los campos de la clase, de este modo se podrán enviar las clases en formato `json`.

```
class Incidencia implements \JsonSerializable
```

```
public function jsonSerialize() {
    return array(
        'id' => $this->getId(),
         'idUsuarioApp' => $this->getIdUsuarioApp(),
    );
}
```

Para ejecutar consultas y obtener información de la base de datos, se usará DQL (Doctrine Query Language). Esto permitirá crear sentencias usando las clases PHP en lugar de tener que usar directamente las tablas de la base de datos. A continuación, se muestra una sentencia DQL:

```
$em = $this->getDoctrine()->getManager();
$dql = "SELECT i FROM CircularBundle\Entity\Incidencia i";
$query = $em->createQuery($dql);
$incidencias = $query->getResult();
```

El acceso a las entidades se hará por medio de un objeto de Doctrine llamado EntityManager, que actúa como administrador de entidades. La primera instrucción del grupo superior sirve para obtenerlo. En las siguientes instrucciones utilizamos el DQL para generar un objeto de Doctrine llamado Doctrine_Query (representado por $query). A este objeto le pedimos que nos devuelva los resultados invocando al getResult(), lo que nos devolverá un array de objetos Incidencia.

1.3. Implementación

Para el desarrollo de este proyecto se utilizará el sistema **MVC** (Modelo - Vista - Controlador). Este patrón de arquitectura de software separa los datos y la lógica de negocio de una aplicación de su representación. Se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

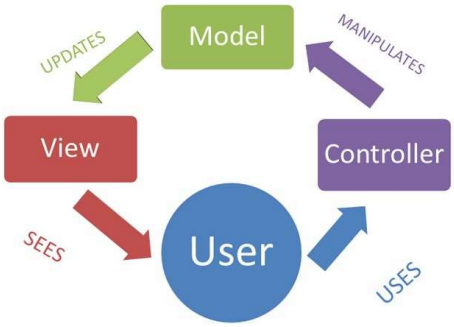

Figura 6: Esquema MVC

1.3.1. Implementación de las interfaces (.html.twig)

Como se ha indicado en el apartado 1.2.2, la interfaz gráfica ha sido proporcionada por el cliente a través de una plantilla con formato html. Lo que se ha hecho para su implementación ha sido adaptar los archivos necesarios al formato .html.twig, y utilizar Twig como motor de plantillas para el proyecto, ya que viene impuesto por Symfony, que lo establece como su motor de plantillas por defecto.

Twig define tres clases de delimitadores:

- {% ... %} para ejecutar declaraciones.
- {{ ... }} para imprimir el contenido de variables.
- {# ... #} se utiliza para añadir comentarios en las plantillas.

1.3.2. Archivos de configuración

En Symfony, para cada funcionalidad de una aplicación, se crea un bundle. Un bundle es simplemente un conjunto estructurado de archivos que se encuentran bajo un mismo directorio y que implementan una característica. En cada directorio se almacena todo lo relacionado con esa característica, incluyendo archivos PHP, plantillas, hojas de estilo, archivos javascript, test y cualquier otro recurso necesario.

Para este proyecto, como sólo se implementará una funcionalidad (gestión de incidencias) se creará un único bundle. Para ello, debemos ir al directorio raíz del proyecto a través de la consola de comandos y ejecutar la siguiente instrucción: `php bin/console generate:bundle --namespace=/CircularBundle --format=yml`. Esto creará la carpeta CircularBundle bajo el directorio /src, con la estructura de directorios que se ve en la figura 7.

Antes de poder utilizar este bundle en la aplicación, es necesario registrarlo. Para ello se debe añadir la instrucción **new** CircularBundle\CircularBundle(), al archivo /app/AppKernel.php dentro de la función registrerBundles().

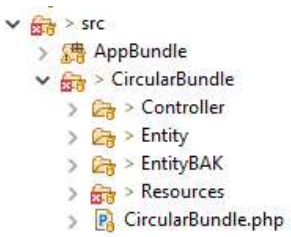

Figura 7: Estructura de directorios

```
public function registerBundles() {
      $bundles = [ new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
           new Symfony\Bundle\SecurityBundle\SecurityBundle(),
           new Symfony\Bundle\TwigBundle\TwigBundle(),
           new Symfony\Bundle\MonologBundle\MonologBundle(),
           new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
           new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
           new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
           new AppBundle\AppBundle(),
           new CircularBundle\CircularBundle(), ];
```

Los archivos de configuración del bundle, **config**, **public** y **views**, se encuentran dentro del directorio Resources. Dentro de **views** están los archivos .html.twig que representan las vistas de la aplicación. En **public** se encuentran los recursos (archivos .css y .js) que utilizan las vistas. La carpeta **config** contiene los archivos routing.yml y webservice.yml, que son los archivos encargados de especificar las rutas de las acciones de la aplicación.

En el archivo routing.yml, las rutas se especifican del siguiente modo:

```
#Ruta para el listado de los organismos
circulapp_organismos_listado:
      path: /organismos
      defaults: { _controller: CircularBundle:Organismos:getOrganismos }
```

La primera línea (`circulapp_organismos_listado`) especifica el `path` que se usará dentro de los archivos de la aplicación (controladores y vistas). La instrucción `path` define la ruta que se mostrará en la `url` al acceder al recurso. La última línea, `defaults`, establece la ruta del `controller` y la `action` dentro de éste que se encargará de gestionar la llamada a ese `path`.

Dentro de este archivo también se define la ruta de enlace al `webservice.yml`, a través del siguiente grupo de instrucciones:

```
#Ruta de enlace al WebService
CircularBundle_webservice:
      resource: "@CircularBundle/Resources/config/webservice.yml"
      prefix:   /api
```

En ellas se indica la ruta del recurso con la instrucción `resource`, y con `prefix` se especifica que todas las rutas definidas dentro del archivo `webservice.yml`, deben ir precedidas de `/api`.

En el archivo `webservice.yml` se definen las rutas que serán usadas por la aplicación móvil. Su estructura es similar a las rutas definidas en el archivo `routing.yml`, salvo que se añade una nueva instrucción, `methods`, la cual especifica qué método deberán implementar las peticiones `HTTP` que utilicen esa ruta.

```
#Ruta para crear una incidencia
crear_incidencia:
      path:  /crearIncidencia
      defaults: { _controller: CircularBundle:WebService:crearIncidencia }
      methods:  [POST]
```

1.3.3. Implementación de la lógica de negocio (`controllers`)

En las aplicaciones `Symfony`, la lógica de negocio se implementa en archivos `php` que reciben el nombre de `controllers`. Como regla general, se crea un controlador para cada entidad de la aplicación. En este proyecto también se han creado dos archivos adicionales: `Auxiliar.php`, en el cual están implementadas funciones que serán usadas por el resto de controladores; y `WebServiceController.php`, archivo que contendrá la lógica de negocio de todas las funcionalidades que se ejecuten desde la aplicación móvil.

A las cabeceras de las funciones definidas en los controladores se les debe añadir el sufijo `Action`. Estas funciones reciben un objeto `Request` y pueden devolver un objeto de tipo `Response` (con formato `json` para enviar datos a la aplicación móvil), o una redirección a vistas `.html.twig`.

Para que estos archivos sean reconocidos por Symfony como controladores, se debe añadir la instrucción `extends Controller` en la definición de la clase.

```
class AdministradorController extends Controller
```

1.3.3.1. Controlador Auxiliar.php

En el controlador `Auxiliar.php` están definidas las funciones para guardar una imagen subida desde la aplicación móvil, guardar una imagen subida desde el panel de administración, y gestionar la paginación del listado de incidencias.

La función `uploadImage` es la encargada de almacenar imágenes subidas desde la aplicación móvil, y se le pasan dos parámetros: uno de tipo `Request`, y el nombre con el que queremos guardar la imagen. Lo primero que se hace es establecer el directorio en el que se va a guardar la imagen, y comprobar que puede editarse. Si no es así, se le dan permisos de escritura con la instrucción `chmod($file_path, 0777)`.

Con la siguiente instrucción, `$request->files->get($name)`, estamos accediendo a la propiedad `files` del atributo `$request`. Esta propiedad es el equivalente en php a `$_FILES`, que es la variable utilizada para la subida de ficheros `HTTP`. Todas las propiedades accesibles a través de `$request` son una instancia de `ParameterBag`, que actúa como un diccionario con pares de elementos clave/valor. En nuestro caso, `files` es una instancia de `FileBag`, que es un contenedor de archivos, de modo que podemos llamar al método `get` con el nombre del archivo que queremos subir, lo que lo convertirá en una variable de tipo `File`.

A esta variable le aplicamos el método `guessExtension()`, que devuelve la terminación del archivo basándose en su tipo `mime`.

El último conjunto de instrucciones se encarga de darle un nombre al fichero y moverlo a la ubicación que se haya determinado.

```
public static function uploadImagen(Request $request, $name){
	$file_path = "uploads/";
	// se comprueba si la carpeta de destino puede editarse
	if (!is_writable($file_path))
		chmod($file_path, 0777);
	// se crea una variable de tipo File
	$file = $request->files->get($name);
	// obtendo la extensión del archivo creado
	$ext = $file->guessExtension();
	// se le pone un nombre al fichero
	$file_name = time() . "." . $ext;
	// se asigna un path completo en el que guardar el archivo
	$file_path = $file_path . $file_name;
	if(move_uploaded_file($_FILES[$name]['tmp_name'], $file_path)) {
		$resultado = $file_name;
	} else{
		$resultado = "";
	}
	return $resultado;
}
```

Para controlar el listado de incidencias, se utiliza la función `getPaginate()`, a la que se le pasan como parámetros un número máximo de incidencias a mostrar por página (`$pageSize`), la página actual en la que estamos (`$currentPage`), una sentencia `dql` con unos parámetros, y un objeto de tipo `doctrine`.

La implementación se realiza creando una `query` con los atributos `dql` y los parámetros enviados. También se le indica cuál debe ser el primer resultado de la sentencia, y el máximo de resultados que debe devolver. Para ejecutar esta sentencia, creamos un objeto de la clase `Paginator` asignándole la `query` que hemos creado. Este objeto contendrá el listado de incidencias tal como hemos establecido en la sentencia.

```
public static function getPaginate($pageSize=10, $currentPage, $dql, $parametros,
      $doctrine){
           $em = $doctrine->getManager();
           $query = $em->createQuery($dql)->setParameters($parametros)
              ->setFirstResult($pageSize * ($currentPage - 1))
              ->setMaxResults($pageSize);
           $paginator = new Paginator($query, $fetchJoinCollection = true);
           return $paginator;
}
```

1.3.3.2. WebServiceController

En este controlador hay implementadas dos tipos de funciones: las que crean objetos en la base de datos (`newIncidenciaAction` y `newUsuarioAppAction`) y la función que devuelve un listado de objetos de la base de datos (`getListadoAction`).

A las funciones que crean objetos se les pasa un parámetro de tipo `Request`, con el cual se recogen los datos de la petición que se le envía. La implementación de estas funciones sigue unos pasos determinados:

1. Se crea un objeto `EntityManager` a través de `doctrine`.
2. Se recogen los datos necesarios para la creación del objeto con el parámetro `$request` y la función `getPost` implementada en `Auxiliar.php`.
3. Se crea el objeto y se le asignan los atributos necesarios.
4. Se almacena el objeto en la base de datos mediante el `EntityManager`, y se libera la información almacenada de éste.

```
public function newUsuarioAppAction (Request $request) {
       $em = $this->getDoctrine()->getManager();
       $email = Auxiliar::getPost('email', $request);
       $pass = Auxiliar::getPost('pass', $request);
       $usuarioApp = new UsuarioApp();
       $usuarioApp->setEmail($email);
       $usuarioApp->setPassword($pass);
       $em->persist($usuarioApp);
       $em->flush();
}
```

Para obtener un listado de objetos de la base de datos, se debe realizar siempre el mismo procedimiento, independientemente de la entidad de la que queramos obtener el listado. Por ello, se ha creado una función en el archivo `Auxiliar.php` (`getListado`) a la que le pasamos el nombre de la clase cuyo listado de objetos queremos recuperar.

El código de esta función consiste en una sentencia `dql`, utilizando el nombre de la entidad que se le ha pasado como parámetro, y la devolución de un objeto tipo `Response` con el listado de objetos obtenido en formato `json`.

1.3.3.3. Controladores genéricos

En los controladores correspondientes a entidades (`AdministradorController`, `IncidenciasController`, `MunicipiosController` y `OrganismosControlles`) se implementan las funciones necesarias para obtener, editar y crear la entidad con la que están relacionados.

Las funciones para obtener y crear ya han sido explicadas. A la función editar (`editProcessAction`), se le pasa como parámetro un objeto de tipo `Request`, y a través de su método `get()`, se obtiene el identificador del objeto que se quiere editar. Para obtener ese objeto, se utiliza el método `getRepository` de `Doctrine`. A este método se le pasa como parámetro una cadena de texto indicando la entidad que se quiere extraer. A continuación, se utiliza la función `findBy` sobre el objeto `repository`, y se especifica el atributo de la entidad por el que se quiere buscar, y el valor de ese atributo.

En la siguiente tabla se muestran las instrucciones descritas para obtener un objeto `Incidencia` a través de su atributo `id`.

```
$repository = $this->getDoctrine()->getRepository(
                          'CircularBundle\Entity\Incidencia');
$incidencia = $repository->findBy(array("id" =>
               $request->request->get("id_incidencia")));
```

1.3.3.4. Implementación del Login

Para gestionar el login de usuarios, hay que seguir una serie de pasos establecidos por `Symfony`. La seguridad de las aplicaciones `Symfony` se establece en el archivo de configuración `app/config/security.yml`. En este archivo se definen distintas secciones para controlar el acceso a la aplicación.

En la sección `encoders` se indican las entidades que mapean tablas cuyas contraseñas están encriptadas. En la base de datos, la contraseña de los administradores se guarda encriptada mediante el algoritmo `sha1`, por lo que indicamos el tipo de encriptación mediante la opción `algorithm`. De este modo, no habrá problemas a la hora de autenticar la contraseña proporcionada por el usuario al acceder al panel.

```
encoders:
     CircularBundle\Entity\Administrador:
          algorithm: sha1
          iterations: 1
          encode_as_base64: false
```

La sección `providers` es la encargada de crear los usuarios de la aplicación. En `Symfony` hay dos tipos de proveedores: `memory` (los usuarios se crean en memoria con los datos incluidos en el propio archivo `security.yml`) y `entity` (los usuarios se crean mediante entidades). Nuestra configuración de la sección `providers` indica que los usuarios se crearán a partir de las entidades `Administrador` y `UsuarioApp`.

```
providers:
     admin:
          entity:
               class: CircularBundle\Entity\Administrador
               property: username
     usuario_app:
          entity:
               class: CircularBundle\Entity\UsuarioApp
               property: nombre
          in_memory:
                 memory: ~
```

La sección `access_control` indica que sólo los usuarios cuyo rol sea `ROLE_ADMIN` podrán acceder a las secciones del panel que empiecen por `/incidencias`.

```
access_control:
     - { path: ^/incidencias, roles: ROLE_ADMIN }
```

En la sección `firewalls` se definen zonas de la aplicación que deben ser protegidas. Las `urls` que coincidan con la expresión regular definida en la opción `pattern` activarán el mecanismo de autenticación asociado al `firewall`. El siguiente `firewall`, al no tener opción `pattern`, protege todas las `urls` de la aplicación. Su mecanismo de autenticación es `form_login`, lo que significa que se mostrará un formulario para introducir un usuario y una contraseña.

```
firewalls:
     main:
          anonymous: ~
          provider: admin
          form_login:
               login_path: /login
               check_path: /login_check
               default_target_path: /incidencias
          logout:
               path: logout
               target: /
```

La opción `login_path` hace referencia a la ruta `/login`. Esta ruta está definida en el archivo `CircularBundle/Resources/config/routing.yml`.

```
#Ruta del Login del panel
login:
     path:     /login
     defaults: { _controller: CircularBundle:Administrador:login }
```

El código anterior indica que al acceder a la ruta `/login`, se ejecutará la función `loginAction` que está definida en el controlador `AdministradorController`. El código de esta función es el siguiente:

```
public function loginAction(Request $request) {
     $sesion = $request->getSession();
     //obtener, si existe, el error producido en el último intento de login
     if ($request->attributes->has(Security::AUTHENTICATION_ERROR)) {
          $error = $request->attributes->get(Security::AUTHENTICATION_ERROR);
     } else {
          $error = $sesion->get(Security::AUTHENTICATION_ERROR);
          $sesion->remove(Security::AUTHENTICATION_ERROR);
     }
     return $this->render(
          'CircularBundle:Default:loginAdmin.html.twig', array(
               'last_username' => $sesion->get(Security::LAST_USERNAME),
               'error' => $error)
          );
}
```

Si ha habido un error en un intento de inicio de sesión, queda guardado en la variable `$error`, y es devuelto a la vista `loginAdmin.html.twig`. En esta vista habrá un formulario, cuyo `action` será `login_check`, y los campos de usuario y contraseña tendrán como atributo `name` `_username` y `_password` respectivamente (es requisito para implementar el login de Symfony). No es necesario crear la acción `login_check` ya que Symfony intercepta el envío del formulario y se encarga de comprobar el usuario y contraseña.

1.3.4. Implementación de la vista del mapa

El desarrollo de esta funcionalidad se ha realizado mediante código `javascript` incluido en la vista destinada a mostrarlo. La clase que controla qué datos se le envían a esta vista es el controlador `MapController`. Este archivo devuelve los datos para generar el mapa a través de la función `getAction`. En este método se evalúa el tipo de administrador logueado en el panel de administración. Si el administrador pertenece al Consorcio de Aguas y Residuos, el centro del mapa se situará en Logroño, y se crearán tantos marcadores como incidencias se hayan almacenado en la base de datos.

Para el resto de los casos en los que el administrador pertenezca a un organismo distinto, primero se obtendrán las incidencias que pertenezcan a ese organismo, y a continuación se buscará el municipio asociado. De este modo, se podrá devolver la posición geográfica de ese municipio, y centrar el mapa sobre esas coordenadas.

El código `javascript` con el que se crea el mapa y los marcadores se implementa en la vista `map.html.twig`. En él se inicializa un nuevo mapa mediante `google.maps.map`, y se establece el centro, el zoom y el tipo de mapa, como se puede ver en la figura 8.

El siguiente grupo de instrucciones itera sobre el array de incidencias enviado a la vista para crear los marcadores asociados a cada una y añadirlos al mapa. A estos marcadores se les asigna un atributo `url`, de modo que al hacer click sobre ellos, se acceda a la vista de edición de la incidencia a la que corresponden.

```
<script type="text/javascript">
    var map;
    function initMap() {
        map = new google.maps.Map(document.getElementById('map'), {
            center: { lat: {{ lat }}, lng: {{ lon }} },
            zoom: 12,
            mapTypeId: google.maps.MapTypeId.ROADMAP
        });
        var marker;
        {% for i in incidencias %}
            marker = new google.maps.Marker({
                position: new google.maps.LatLng(
                    {{ i.getLatitud() }},
                    {{ i.getLongitud() }}
                ),
                map: map,
                title: '{{ i.getTitulo() }}',
                url: '{{ path('circulapp_incidencia_edit', { 'id':
                    i.getId() }) }}'
            });
            marker.addListener('click', function() {
                window.location.href = marker.url;
            });
       {% endfor %}
    }
</script>
```

Figura 8: Código javascript para la creación del mapa

1.4. Herramientas utilizadas

- **API egeloen/google-map**

Al empezar a realizar la implementación del mapa, se buscó la forma de hacerlo mediante herramientas específicas de Symfony. La API egeloen/google-map permite crear un mapa desde un controlador, y enviarlo a una vista como una variable más. Sin embargo, la versión de composer que se ha utilizado para el proyecto es la 1.6.4, y esta API no esta implementada para esta versión.

- **Entorno de desarrollo integrado**

Como IDE para el desarrollo de la aplicación web se ha utilizado Eclipse, en su versión para desarrolladores php.

- **Lenguaje de implementación**

El lenguaje elegido para el desarrollo del panel web ha sido php, ya que al elegir el framework Symfony, era un requisito obligado.

1.5. Pruebas

En la siguiente tabla se indican las pruebas realizadas y el resultado obtenido.

Prueba	Resultado
Inicio de sesión	Correcto
Editar incidencia	
Cambiar estado	Correcto
Añadir comentario	Correcto
Modificar imagen	Incorrecto
Crear organismo	Correcto
Acceder al detalle de una incidencia a través del mapa	Correcto

Modificar imagen

Al acceder a la vista de edición de una incidencia, si no se seleccionaba una nueva imagen para la incidencia y se dejaba el valor por defecto, la imagen devuelta era nula. Esto sucedía porque los campos encargados de mostrar y seleccionar imagen eran distintos. Para solucionar este error, se ha introducido una condición en el método `editProcessAction` que comprueba si se ha elegido una imagen. Si no se ha elegido, no se sobrescribe la imagen de la incidencia.

```
//verificamos si se ha elegido imagen
if (!empty($request->files->get('imagen'))) {
   reset($incidencia)->setImagen(Auxiliar::postFichero('imagen', $request));}
```

2. Subproyecto 2: Aplicación iOS

2.1. Análisis

Esta parte del proyecto consistirá en el desarrollo de una aplicación para dispositivos `iOS`, que permita a sus usuarios enviar información sobre los desperfectos que encuentren en los contenedores de recogida de residuos de su municipio.

Como se ha indicado en el apartado 1.1, el proceso de captura de requisitos se ha llevado a cabo a través de reuniones con el cliente (véase anexo 1).

2.1.1. Análisis de los requisitos funcionales

1. La aplicación mostrará un mensaje de bienvenida al usuario cada vez que la inicie.

2. El usuario deberá loguearse en la aplicación para reportar una incidencia.

3. El usuario podrá generar incidencias, para las cuales deberá especificar:

    - un título para la incidencia
    - una imagen descriptiva de la incidencia
    - una descripción para la incidencia ya sea de forma escrita o a través de un mensaje de audio
    - una categoría para la incidencia

4. El usuario podrá elegir entre una de las siguientes categorías para clasificar la incidencia:

    - Desperfectos visuales (contenedores pintados, quemados o abollados)
    - Contenedores desbordados o llenos
    - Contenedores sucios, en mal estado o inutilizables
    - Ausencia o lejanía de contenedores
    - Malos olores, provocados por los contenedores

5. Si el usuario ha optado por enviar la descripción de la incidencia en forma de mensaje de audio, la aplicación será capaz de convertir ese audio en información textual.

6. El usuario podrá, una vez rellenados todos los campos necesarios para crear una incidencia, enviar la incidencia en ese momento, o guardarla en el dispositivo y enviarla posteriormente.

7. El usuario podrá enviar las incidencias que genere, siempre que haya establecido la información arriba especificada.

2.1.2. Análisis de los requisitos no funcionales

1. La aplicación deberá ser desarrollada para dispositivos iOS.

2. La interfaz de la aplicación será simple y seguirá el estilo y colores propios de Ecoembes y *TheCircularLab*. Se visualizará correctamente en dispositivos iPhone 6, iPhone 7 y iPhone 8. Adicionalmente, es deseable que también se adapte a las pantallas de los iPads.

2.1.3. Tecnología a utilizar

Como IDE para desarrollar la aplicación iOS será imprescindible utilizar Xcode. Este IDE contiene un conjunto de herramientas creadas por Apple destinadas al desarrollo de software para macOS, iOS, watchOS y tvOS.

El lenguaje de programación elegido para el desarrollo de la aplicación iOS será Swift. Este es un lenguaje de programación multiparadigma creado por Apple enfocado al desarrollo de aplicaciones para iOS y macOS. La principal razón de desarrollar el proyecto con Swift en lugar de Objective-C es que ha sido un requisito impuesto por la empresa, ya que quiere usarlo como base para el desarrollo de una aplicación en la que implementar todas las funcionalidades que desean.

Swift presenta varias ventajas frente a Objective-C. El código es más fácil de mantener, ya que mientras que en Objective-C hay dos archivos para cada clase (archivo de cabeceras .h y archivo de implementación .m), Swift los combina en un único archivo de código (.swift). Desarrollar en Swift es también más seguro. En Objective-C, se puede llamar a un método con un puntero sin inicializar (nulo), y la ejecución del programa no se detendría. En Swift, se utiliza un tipo especial de datos, optionals, los cuales validan si una variable ha sido inicializada o no antes de usarse.

2.1.4. Alcance del proyecto

1. **Mantenimiento**

Al concluir el proyecto y entregar la aplicación al cliente, su posterior mantenimiento será responsabilidad de *TheCircularLab*.

2. **Despliegue**

La aplicación a desarrollar no será una aplicación completa desde el punto de vista del cliente, es decir, solo se implementará una funcionalidad para obtener un producto mínimo viable. Por esto, es probable que la aplicación sufra cambios una vez finalizado el proyecto, cambios que serán llevados a cabo por el cliente, *TheCircularLab.*

La posterior distribución en la AppStore también será llevada a cabo por el cliente.

3. **Manual de usuario**

Como ya se ha indicado, para la aplicación solo se implementará una funcionalidad, por lo que no será necesario la creación de una guía o manual de usuario, al ser una aplicación sencilla, sin muchas secciones.

2.1.5. Planificación

La planificación de la aplicación iOS ha sido explicada junto con la planificación del panel de administración web, en el apartado 1.1.5 del subproyecto 1.

El total de horas estimadas para desarrollar esta aplicación serán 160 horas. Se destinarán más horas al desarrollo de este proyecto que al del panel de administración ya que no se cuenta con experiencia desarrollando una aplicación para dispositivos iOS.

2.1.6. Plan de pruebas

A continuación, se indican las pruebas que se realizarán una vez implementada la funcionalidad del proyecto.

- Inicio de sesión
- Registro
- Crear y enviar una incidencia
- Crear y guardar una incidencia
- Enviar una incidencia guardada
- Ver el listado de incidencias

En las pruebas de **Inicio de sesión** y **Registro**, se probarán todas las posibilidades a la hora de introducir la información requerida (campos vacíos, contraseñas que no coincidan, etc.) para verificar que ha sido implementado correctamente.

En las pruebas en las que se **cree una incidencia**, se probará que se han rellenado todos los campos que se piden, y que, si no es así, no se puede crear la incidencia.

En la prueba de **enviar una incidencia guardada**, se probará que la acción de enviar sólo estará disponible para incidencias que no hayan sido ya enviadas. También se verificará que, una vez enviada la incidencia, su estado cambie de *guardada* a *enviada*.

En el **listado de incidencias**, se debe comprobar que las incidencias mostradas han sido creadas por el usuario que ha iniciado sesión, y que no aparecen incidencias duplicadas (por haber estado primero almacenadas en el dispositivo y luego haberlas enviado).

2.2. Diseño

2.2.1. Diseño de interfaces

El diseño de las distintas interfaces de la aplicación se ha realizado siguiendo la guía de estilo proporcionada por la empresa. Este documento está disponible en el anexo 4.

2.2.1.1. Prototipos iniciales

Los prototipos de las interfaces han sido realizados con la herramienta online `Proto.io`. Se ha elegido por permitir crear interfaces específicas para `iOS`, usando sus elementos y componentes propios para la creación de prototipos funcionales.

El diseño de estos prototipos puede verse en el anexo 3.

2.2.1.2. Diagrama de navegabilidad inicial

En el diagrama de navegabilidad se indica el modo en el que el usuario interactuará con la aplicación, teniendo como base las interfaces creadas para la realización de los prototipos.

La navegabilidad de la aplicación puede verse en el anexo 3.

2.2.1.3. Diseño final de interfaces

Durante el desarrollo del proyecto se han mantenido diversas reuniones con el cliente, con el objetivo de mostrarle el avance de éste. Como consecuencia surgieron diversos cambios, algunos de ellos relativos a las interfaces de la aplicación. Estos cambios provocaron la inclusión en el proyecto de tres nuevas interfaces: un **login** para los usuarios; un **registro**; y una vista de **detalle de incidencias**.

A continuación, se indican qué interfaces han sufrido cambios después de las reuniones con el cliente, así como las nuevas interfaces que se han debido incluir. Las interfaces correspondientes a la pantalla de bienvenida y configuración no han sufrido modificaciones.

1. **Pantalla de Login (figura 9)**

Interfaz diseñada para actuar como inicio de sesión. Se compone del logotipo de la empresa, dos campos para que el usuario introduzca su email y su contraseña, y un enlace para ir a la pantalla de registro en caso de que el usuario no esté registrado. Esta interfaz cumple el requisito funcional 2.

Figura 9: Interfaz de login

2. **Pantalla de Registro (figura 10)**

Esta interfaz es muy similar a la de login. En ella se le piden al usuario los datos necesarios para darlo de alta en la aplicación: email, contraseña y repetir contraseña (para asegurarnos que no ha habido error en su introducción). También cuenta con un enlace para regresar a la pantalla de login.

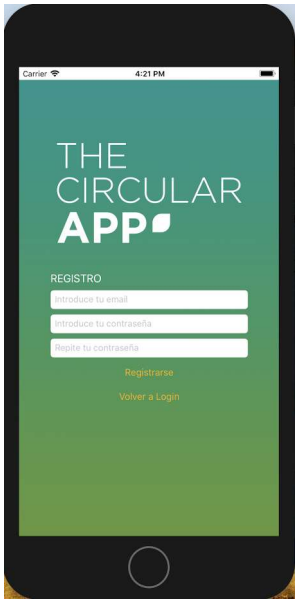

Figura 10: Interfaz de registro

3. Pantalla Nueva Incidencia (figura 11)

Los cambios en esta interfaz han sido la inclusión del icono del micrófono para grabar la descripción proporcionada por el usuario, y la imagen en la parte inferior de la pantalla, que sustituye al símbolo "+" para seleccionar una imagen para la incidencia. Esta interfaz está relacionada con los requisitos funcionales 3, 4, 5 y 6.

Figura 11: Interfaz Nueva incidencia

4. Pantalla Detalle incidencia (figura 12)

Esta interfaz no estaba contemplada inicialmente como una de las vistas de la aplicación. En ella se muestra el título, descripción, categoría, imagen y estado de la incidencia. Si el estado no ha cambiado de su valor inicial (PENDIENTE), el estado será *en trámite*. Si su estado ha cambiado, se indicará su nuevo valor, así como el comentario indicado por el administrador que ha gestionado la incidencia. Esta interfaz cumple con el requisito funcional 7.

Figura 12: Interfaz detalle de incidencia

2.2.1.4. Diagrama de navegabilidad final

Debido a la creación de nuevas vistas para la aplicación, el diagrama de navegabilidad sufrió cambios para incluirlas. En la figura 13 se muestran las interacciones que puede realizar un usuario al ejecutar la aplicación.

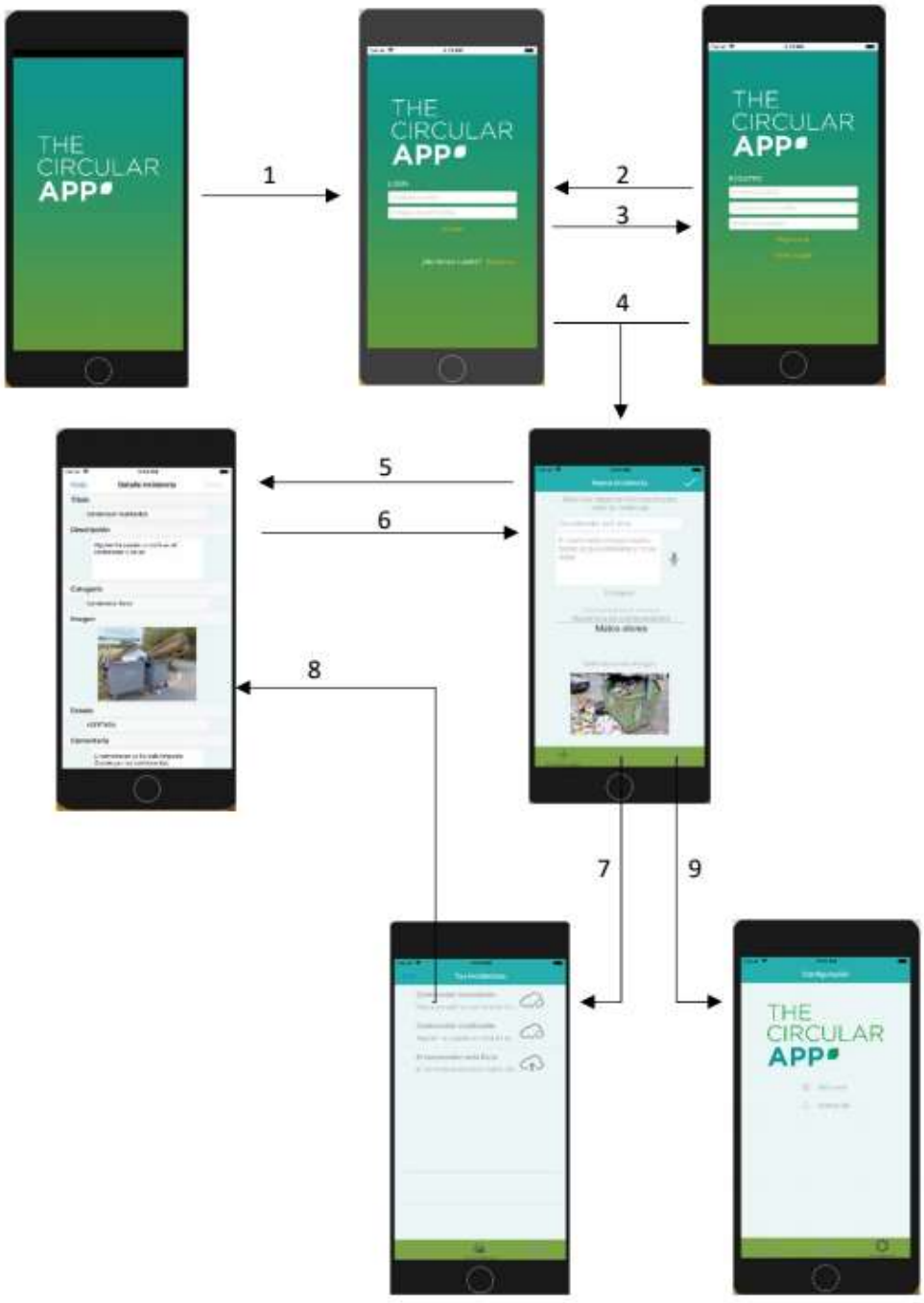

Figura 13: Diagrama de navegabilidad final

Al inicio de la aplicación, se muestra la pantalla de bienvenida. A continuación, se ejecuta la secuencia 1 y se lleva al usuario a la pantalla de login. En esta pantalla, el usuario introducir sus datos y acceder a la aplicación (4), o ir a la pantalla de registro (3). Desde la interfaz de registro, el usuario puede volver al login (2) o acceder a la aplicación (4). Una vez en la pantalla de nueva incidencia (4), el usuario podrá crearla y acceder a la vista de detalle de la incidencia (5). Desde esta vista podrá volver a la interfaz de nueva incidencia pulsando el botón de la barra de navegación *Atrás*

(6). En las pantallas de Nueva incidencia, Listado y Configuración¸ el usuario dispone de una barra de navegación inferior. Este elemento permite al usuario acceder a las pantallas de Nueva incidencia, Listado y Configuración. En la interfaz de Listado, cuando el usuario seleccione una incidencia, se le mostrará el detalle de ésta (8).

2.2.2. Diseño de clases

Las clases creadas para aplicación pertenecen a dos grupos distintos. Uno de ellos las componen las clases que mapean entidades de la base de datos (`Incidencia`, `UsuariosApp`, `Organismo` y `Categoria`) y el otro grupo lo forman las clases creadas para manejar las vistas de la aplicación (`ConfiguracionViewController`, `EditIncidenciaTableViewController`, `IncidenciaTableViewCell`, `IncidenciaTableViewController`, `LoginViewController`, `NuevaIncidenciaViewController`, `PantallaInicioViewController`, `RegistroViewController`).

Las clases que mapean entidades de la base de datos están formadas por un constructor para inicializar objetos de esa clase, excepto la clase `Incidencia`, a la que se le han añadido métodos para hacer que sus objetos sean almacenables en el dispositivo, como se explica en el apartado de implementación.

El proyecto también cuenta con una clase presente en todos los proyectos `Swift`, `AppDelegate`. Este archivo controla el ciclo de vida de la aplicación, y contiene métodos que responden ante los cambios de estado de ésta.

El conjunto de interfaces de la aplicación lo componen vistas (`UIView`), barras de navegación (`UINavigation`), barras de estado (`UITabBar`), tablas (`UITableView`) y celdas de las tablas (`UITableViewCell`). Para cada uno de ellos ha sido necesario crear una clase de tipo `cocoa class`, la cual debe implementar el tipo del elemento al que se quería asociar.

Las clases que se vayan a asociar a vistas (`UIView`), deben implementar la interfaz `UIViewController`. La clase `IncidenciaTableViewController` se encargará de gestionar el listado de incidencias, y debe implementar la interfaz `UITableViewController`. Para las celdas del listado, se ha creado la clase `IncidenciaTableViewCell`, la cual hereda de `UITableViewCell`.

Para que las interfaces estén vinculadas a las clases que hemos creado, se debe modificar su atributo `Class` y especificar la clase que se desee. Este atributo se encuentra dentro del conjunto de propiedades del `Identity Inspector` de cada interfaz.

En la figura 14 se pueden ver las distintas clases que componen el proyecto y con qué interfaces están vinculadas.

Figura 14: Clases relacionadas con las interfaces

2.3. Implementación

2.3.1. Implementación de las interfaces

La implementación de las interfaces se ha realizado dentro del archivo `Main.storyboard`. Este archivo es la representación visual de las interfaces de la aplicación, en el que se muestran las pantallas y las conexiones entre esas pantallas.

Las pantallas de bienvenida, login, y registro solo están compuestas por un elemento `UIView`. Este elemento actúa como un simple contenedor en el que situar los componentes que formarán la vista.

El resto de las pantallas de la aplicación están compuestas por tres elementos distintos: la vista propia de la pantalla que actuará como contenedor de sus elementos; un elemento `Navigation View`; y un elemento `Tab Bar View`. En la figura 15 se explica cómo se componen las pantallas con estos elementos.

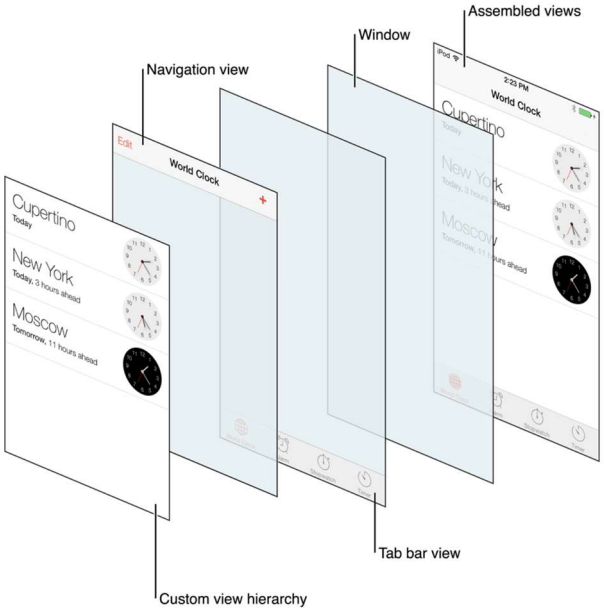

Figura 15: Composición vistas iOS

2.3.2. Archivo Info.plist

Este archivo es un listado con las propiedades de la aplicación, y contiene información sobre la configuración de ésta. El contenido de este archivo está estructurado en `xml` y está compuesto de pares clave - valor. El sistema utiliza estas claves para obtener información de la aplicación y cómo está configurada.

Para este proyecto ha sido necesario añadir las siguientes instrucciones al archivo `Info.plist`:

```
<key>NSPhotoLibraryUsageDescription</key>
    <string>This application will use your photos</string>
<key>NSLocationWhenInUseUsageDescription</key>
    <string>This application will use location to upload information.</string>
```

La primera pareja clave - valor ha sido necesaria para permitir acceder a la galería de imágenes del usuario. La segunda, para obtener la localización del dispositivo. La primera vez que el usuario ejecute la aplicación se le mostrará un mensaje preguntando por las funcionalidades que se han añadido (`NSPhotoLibraryUsageDescription` y `NSLocationWhenInUseUsageDescription`) para que el usuario decida permitirlas o no.

2.3.2. Implementación de la lógica de negocio

En este apartado se explicarán los archivos y funciones más importantes para la implementación de la funcionalidad del proyecto

2.3.3.1. AppDelegate.swift

Este archivo controla el ciclo de vida de la aplicación, y contiene los métodos que responden ante los cambios de estado de ésta.

El método `application` contiene el código a ejecutar al iniciar la aplicación. Se ha implementado una llamada asíncrona a la función del servidor `getIncidencias()` dentro de este método, para que se carguen las incidencias almacenadas en la base de datos sin que el usuario tenga que esperar para continuar usando la aplicación.

Para ello se utiliza `URLSession` para crear una petición `HTTP` tipo `GET`:

```
let url = URL(string: url_servidor + "getIncidencias")
URLSession.shared.dataTask(with:url!, completionHandler: {
        (data, response, error)
```

Como el método `getIncidencias()` devuelve una respuesta en formato `JSON`, se debe parsear la respuesta para formar un array de incidencias. De este modo se podrá iterar sobre el array y crear incidencias a partir de los datos obtenidos:

```
let json = try JSONSerialization.jsonObject(with: data, options:
           .allowFragments) as! [String : AnyObject]
let incidencias_array = json["incidencias"] as? [[String : AnyObject]] ?? []
```

Del mismo modo, se han implementado llamadas a los métodos del servidor `getUsuariosApp`, `getOrganismos` y `getCategorias`, todos realizados de manera asíncrona, para permitir que la aplicación continúe su ejecución sin tener que esperar a recibir la respuesta.

Como en la aplicación se van a manejar incidencias almacenadas en la base de datos y en el dispositivo, se han creado dos array de incidencias (`incidencias_bbdd` e `incidencias_dispositivo`) para saber si una incidencia ha sido enviada o no. Estas variables han sido definidas en este archivo, de modo que serán accesibles por toda la aplicación.

2.3.3.2. Implementación pantalla inicial

Como se ha explicado, la pantalla inicial será una interfaz de tipo `UIView`. Esta vista se mostrará durante dos segundos y luego dará paso a la interfaz de login. Para implementar este comportamiento, se ha creado una clase que extiende la interfaz `UIViewController`, y se le ha asignado como clase a la interfaz que representa la pantalla inicial en el archivo `Main.storyboard`.

La funcionalidad de está interfaz se ha implementado dentro del método `viewDidLoad`. Este método es común a todas las clases que extienden `UIViewController`, y se ejecuta la primera vez que la interfaz es mostrada en la aplicación. Se ha utilizado el método `scheduledTimer` del objeto `Timer` para hacer que tras dos segundos se ejecute la función `ocultar`. Esta función dispara la realización de la transición (`segue`) `ocultarPantallaInicio`, la cual se ha creado en el archivo `Main.storyboard`. Los métodos `viewDidLoad` y `ocultar` quedan del siguiente modo:

```
override func viewDidLoad() {
    super.viewDidLoad()
    Timer.scheduledTimer(timeInterval: 2.0, target: self,
           selector: #selector(ocultar), userInfo: nil, repeats: false)
}

@objc func ocultar() {
       self.performSegue(withIdentifier: "ocultarPantallaInicio",
             sender: self)
}
```

2.3.3.3. Implementación Nueva incidencia

Los elementos que componen esta interfaz están dispuestos de manera vertical. Por ello, para asegurarse de que el usuario pueda acceder a todos ellos aunque la pantalla de su dispositivo no pueda mostrarlos todos a la vez, se han creado dentro de un `ScrollView`, que actuará como marco en el que se dispondrán los componentes de la vista.

Para poder tener acceso a estos controles, se ha implementado la clase `NuevaIncidenciaViewController`, y se ha establecido que el atributo `Class` de la interfaz sea esta nueva clase. Ahora se podrán manejar los objetos de esta pantalla creando referencias a ellos (`outlets`) en la clase.

En `Swift`, la funcionalidad del objeto `ImageView` es mostrar información (imágenes), por lo que no se puede crear un `action` de este objeto a la clase asociada. Para implementar esta funcionalidad, se debe añadir un objeto del tipo `TapGestureRecognizer` sobre el objeto `ImageView`.

Este tipo de objetos se pueden aplicar a cualquier vista, es decir, objeto de tipo `UIView` o que extienda de ésta. Permite controlar el número de veces que se pulsa el objeto al que están asociados, si se hace con uno o más dedos, si se arrastra el objeto, o si es una pulsación prolongada.

Una vez añadido este objeto sobre la imagen, aparecerá en la parte superior de la interfaz, como se puede ver en la figura 16. Ahora ya podremos crear un `action` desde el objeto `TapGestureRecognizer` a la clase `NuevaIncidenciaViewController`.

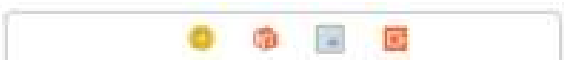

Figura 16: TapGestureRecognizer

Para poder trabajar con imágenes y seleccionarlas, en la declaración de la clase se ha añadido `UIImagePickerControllerDelegate.` Esta interfaz está formada por una serie de métodos que nuestra clase debe implementar para poder trabajar con un selector de imágenes.

El método `selectImage` es el que se ejecuta cuando se pulsa sobre la imagen, es decir, es el método asociado al objeto `TapGestureRecognizer` que está vinculado a nuestro objeto `ImageView`. En este método, lo primero que se hace es crear una constante de tipo `UIImagePickerController` y hacer que su `delegate` (controlador) sea la propia clase. A continuación se establece que su origen de datos (imágenes en este caso) sea la galería del dispositivo (`.photoLibrary`). Por último, se llama al método `present` de la clase para que muestre al usuario las imágenes de la galería.

```
@IBAction func selectImage(_ sender: UITapGestureRecognizer) {
       let imagePickerController = UIImagePickerController()
      imagePickerController.delegate = self
       imagePickerController.sourceType = .photoLibrary
       present(imagePickerController, animated: true, completion: nil)
}
```

Para controlar la acción `Cancelar` sobre la elección de la imagen, se implementa el método `imagePickerControllerDidCancel`. El objetivo de esta función es llamar al método `dismiss`, lo que hace que la galería se cierre, llevando al usuario de vuelta a la vista.

El tercer método que interviene en la elección de la imagen es `imagePickerController`. Su función es comprobar si se ha elegido una imagen, y si es así, asignarla al atributo `image` del objeto `ImageView` de nuestra vista.

Mediante la función `crearIncidencia`, vinculada al objeto `BarButtonItem` de la barra de herramientas, el usuario puede enviar o almacenar una incidencia.

Lo primero que se comprueba en este método es si el usuario ha introducido toda la información necesaria, mostrándole un aviso si se ha dejado algún campo. Una vez comprobado, se verificará que el usuario ha dado su consentimiento para que la aplicación pueda acceder a su geolocalización, mostrándole un mensaje requiriéndolo en caso contrario. A continuación, se crea un objeto de tipo `CCLocation`, el cual servirá para obtener la latitud y la longitud en la que se encuentre el usuario.

El siguiente paso es presentar al usuario un cuadro de diálogo (`UIAlertController`) dándole la opción de enviar o almacenar la incidencia.

- **Enviar incidencia**

Si se elige enviar, se realizará una petición a la aplicación web cuyo método será `POST` y cuyo contenido será de tipo `multipart/form-data`, ya que se debe enviar la imagen de la incidencia. Para realizar esta petición se ha utilizado la API `Alamofire`, una librería externa para realizar peticiones `HTTP` en `swift`.

En la petición, primero se itera el diccionario de parámetros formado por los datos que el usuario ha introducido para la incidencia. Este diccionario es de tipo `[String : Any]`. En el bucle, cada entrada del diccionario será añadida a la petición con la siguiente instrucción:

```
multipartFormData.append("\(value)".data(using: String.Encoding.utf8)!,
       withName: key as String)
```

En ella se indica que debe coger el valor (`value`) de cada elemento, convertirlo a `String` y añadirlo a la petición con la clave (`key`) indicada.

Para añadir la imagen, primero se debe convertir al tipo `Data`. Para ello se ha utilizado el método `UIImageJPEGRepresentation`, al que se le pasan dos parámetros: uno de tipo `UIImage` y otro de tipo `Float`. El primer parámetro será la imagen seleccionada por el usuario. El segundo representa la calidad que tendrá la imagen `JPEG` resultante, cuyo valor será entre 0.0 (mayor compresión, menor calidad) y 1.0 (menor compresión, mayor calidad).

El siguiente paso es añadirla a la petición. Para ello se utiliza el método append, al que se le indica el dato a añadir (imageData), el nombre con el que se enviará (image), el nombre del archivo (image.jpg), y su tipo mime (image/jpg).

```
let imageData = UIImageJPEGRepresentation(image, 0.5)
multipartFormData.append(imageData!, withName: "image",
          fileName: "image.jpg", mimeType: "image/jpg")
```

Todos los datos de la petición deben codificarse antes de enviarlos. Para ello, se emplea usingThreshold: UInt64.init(). Esto permite a la librería Alamofire determinar si los datos deben ser almacenados en la memoria o en el disco durante el proceso de codificación.

El último paso es indicar a la petición la url (to: url), el método (method: .post), y las cabeceras (headers: headers), y comprobar si la petición ha sido realizada con éxito.

```
Alamofire.upload(multipartFormData: {
    (multipartFormData) in
        for (key, value) in parameters {
            multipartFormData.append("\(value)".data(using:
                String.Encoding.utf8)!, withName: key as String)
        }
        multipartFormData.append(imageData!, withName: "image", fileName:
            "image.jpg", mimeType: "image/jpg")
    }, usingThreshold: UInt64.init(),
       to: url,
       method: .post,
       headers: headers) { (result) in
           switch result{
               case .success(let upload, _, _):
                   // código en caso de éxito
               case .failure(let error):
                   // código en caso de fallo
       }
}
```

- **Guardar incidencia**

Para hacer que una incidencia pueda ser almacenable en el dispositivo, primero se ha debe hacer que la clase Incidencia sea codificable, y luego crear un objeto de esta clase.

Para implementar la codificación en la clase Incidencia, se debe hacer que herede de NSCoding y NSObject.

NSCoding obliga a las clases a implementar dos métodos: encode(with:) e init(coder:), a los que se les pasa como parámetro un objeto de tipo NSCoder. Este objeto será el encargado de invocar a las funciones encode y decode sobre los atributos de la clase. En el método encode(with:) se utiliza NSCoder para codificar esos atributos, y en init(coder:) para inicializarlos a partir de los datos obtenido con él.

Por último, se han creado los métodos `saveToFile` y `loadFromFile` para guardar y recuperar un array de incidencias. Estas funciones utilizan `unarchiveObject` y `archiveRootObject` de la clase `NSKeyedArchiver`, indicando la ruta en la que se almacena el array.

```
static func loadFromFile() -> [Incidencia]?  {
    return NSKeyedUnarchiver.unarchiveObject(
        withFile: Incidencia.ArchiveURL.path) as? [Incidencia]
}

static func saveToFile(incidencias: [Incidencia]) {
    NSKeyedArchiver.archiveRootObject(incidencias,
        toFile: Incidencia.ArchiveURL.path)
}
```

Una vez implementados los métodos necesarios para hacer que las instancias de la clase Incidencia sean almacenables en el dispositivo, creamos una incidencia con los datos introducidos por el usuario. Esta incidencia se añade al array `incidencias_dispositivo`, y se almacena en el sistema a través de `Incidencia.saveToFile(incidencias: incidencias_dispositivo)`.

- **Reconocimiento de voz para la descripción**

El campo de descripción para la incidencia lo representa un área de texto. Esta información puede incluirse escribiéndola o a través de un audio, que será convertido en texto. Para esta última opción es necesario usar la clase `SFSpeechRecognizer`.

Lo primero que se debe hacer es importar la librería `Speech` (`import Speech`) y hacer que la clase herede `SFSpeechRecognizerDelegate`. Al igual que con la localización GPS y la utilización de la galería de imágenes, se deben configurar los permisos necesarios en el archivo `Info.plist` para que el usuario permita a la aplicación hacer uso del micrófono.

```
<key>NSSpeechRecognitionUsageDescription</key>
    <string>Your speech will be detected</string>
<key>NSMicrophoneUsageDescription</key>
    <string>Microphone is used to capture your speech</string>
```

Para implementar el reconocimiento de voz, se necesitan declarar cuatro variables.

- Una instancia de la clase `AVAudioEngine`, que se encargará de procesar el stream de audio.
- Una instancia de `SFSpeechRecognizer`, a la que se le deberá indicar el idioma que tendrá que identificar (`SFSpeechRecognizer(locale: Locale.init(identifier: "es_ES"))`).
- Una instancia de `SFSpeechAudioBufferRecognitionRequest`, la cual controlará el buffer que se creará cuando el usuario esté usando el micrófono.
- Una instancia de `SFSpeechRecognitionTask`, que gestionará el inicio y la pausa del reconocimiento.

Para administrar este reconocimiento de voz, se ha creado una función que no recibe ningún parámetro, y que será llamada desde la `action` vinculada a la imagen del micrófono en la interfaz. Lo primero que se hará en este método será inicializar el `audioEngine`.

```
guard let node = audioEngine?.inputNode else { return }
let recordingFormat = node.outputFormat(forBus: 0)
node.installTap(onBus: 0, bufferSize: 1024, format: recordingFormat) {
    buffer, _ in request.append(buffer) }
```

Con `inputNode` se crea un `singleton` para recibir la entrada de audio. Con la segunda instrucción establecemos un formato de salida sobre el `bus` 0, y con la tercera, indicamos que `node` debe situarse sobre el `bus` 0. De este modo se ha establecido que este `bus` será a través del cual grabaremos en audio.

A continuación, se ejecuta una llamada a las funciones `prepare()` and `start()` sobre `audioEngine`, para iniciar la grabación de voz. Antes de ejecutar el método `recognitionTask` sobre `speechRecognizer`, comprobamos que está disponible para el dispositivo y para el idioma elegido. Por último, ejecutamos el método y enviamos el resultado al área de texto correspondiente.

```
recognitionTask = (speechRecognizer?.recognitionTask(with: request,
    resultHandler: { result, error in
        if let result = result {
            self.txtDesc.text = result.bestTranscription.formattedString
        } else if let error = error {
             print(error)
        }
}))!
```

2.3.3.4. Implementación del listado de incidencias

En la implementación del listado intervienen dos clases, `IncidenciaTableViewController` e `IncidenciaTableViewCell`. La primera representa la tabla sobre la que se mostrarán las celdas correspondientes a las incidencias. En la segunda clase se indica cómo debe rellenarse la información de la celda.

En `IncidenciaTableViewController`, para mostrar las incidencias del usuario, se ha modificado el método `viewDidAppear`, el cual se ejecuta cada vez que la vista asociada a la clase va a ser mostrada. En este método se comprueba si el usuario ha creado alguna incidencia más desde la última vez que se accedió al listado, y en ese caso, se obtienen de nuevo las incidencias de ese usuario, se vuelcan en la tabla, y se recargan los datos de la tabla (`self.tableView.reloadData()`).

Para que las celdas de la tabla se controlen mediante `IncidenciaTableViewCell`, se ha tenido que indicar que el tipo de estas celdas sea el de esa clase. Para poder elegir qué información mostrar en cada celda, se debe establecer el estilo de la celda a `Custom`. De este modo se podrá elegir qué campos van a componer celda.

En el presente proyecto, la información que se quiere mostrar de la incidencia en el listado es el título, el inicio de la descripción, y un icono, que indica si la incidencia ha sido enviada o no (figura 17). Por ello, las celdas las formarán dos elementos `Text View` situados en la parte derecha de la celda, uno encima de otro, y un elemento `Image View`, en el que se situará el icono.

Figura 17: Icono de incidencia guardada (izq) / Icono de incidencia enviada (der)

En la clase `IncidenciaTableViewController`, el método `cellForRowAt` devuelve un tipo `UITableViewCell`, que corresponderá con el tipo de celda que hemos definido. Para ello, se debe crear una variable `cell`, que representará a cada celda obtenida a través del índice de la tabla (`indexPath`), y que se convertirá al tipo `IncidenciaTableViewCell`. También es necesario obtener la incidencia correspondiente a ese índice, para enviarla al método update de la celda, y establecer así la información de la incidencia en la celda.

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
    IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
             "IncidenciaCell", for: indexPath) as! IncidenciaTableViewCell
        let incidencia = incidencias_user[indexPath.row]
        cell.update(with: incidencia)
         cell.showsReorderControl = true
         return cell
}
```

El usuario también puede eliminar una incidencia que aún no se haya enviado. Para ello, en la función `editingStyle`, se comprueba si la opción elegida ha sido `.delete` y si el atributo `enviada` de la incidencia es false, en cuyo caso, se podrá eliminar. Si no, se le mostrará al usuario un mensaje indicándole que no puede eliminar una incidencia ya enviada.

Cuando se selecciona una incidencia en el listado, se envía al usuario a la vista del detalle de la incidencia. Esta relación tiene como identificador `EditIncidencia`. A través del método `prepareForSegue`, se verifica que la relación es `EditIncidencia`, y se obtiene la incidencia que ha sido seleccionada. Se crea una instancia de la clase con la que se relaciona la interfaz del detalle de la incidencia (`EditIncidenciaTableViewController`), y se le asigna la incidencia seleccionada, para que sus datos sean accesibles desde la clase.

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "EditIncidencia" {
        let indexPath = tableView.indexPathForSelectedRow!
        let incidencia = incidencias_user[indexPath.row]
        let nav = segue.destination as! UINavigationController
        let editIncidenciaTableViewController = nav.topViewController as!
            EditIncidenciaTableViewController
        editIncidenciaTableViewController.incidencia = incidencia
    }
}
```

2.3.3.5. Implementación detalle de una incidencia

En la vista del detalle de una incidencia, se van a mostrar el título, la descripción, la categoría, la imagen y el estado en el que se encuentra la incidencia. Si la incidencia ya ha sido gestionada (su estado ha cambiado a *ACEPTADA* o *RECHAZADA*), también se mostrará el comentario añadido por el administrador explicando la razón de su cambio de estado. De este modo, el usuario podrá obtener un feedback sobre sus incidencias.

Para las incidencias guardadas en el dispositivo que aún no hayan sido enviadas, el usuario dispondrá de un botón a través del cual enviarlas. Al hacerlo, la incidencia se eliminará del dispositivo, para lo cual se debe eliminar primero del array que representa a las incidencias

guardadas, y luego volver a almacenar el array mediante `Incidencia.saveToFile(incidencias: incidencias_dispositivo)`.

2.3.3.6. Implementación pantalla de configuración

La pantalla de configuración se compone de dos etiquetas. Al igual que con la imagen en la interfaz `NuevaIncidencia`, para permitir que el usuario pueda seleccionar la etiqueta, se ha debido añadir un `TapGestureRecognizer` sobre ella.

Para abrir la página web de la empresa, se debe crear una variable de la clase `URL`, a la que se le pasa como parámetro en el constructor una cadena que representa la `url` de la dirección web que queremos abrir. A continuación, se llama al método `open` de `UIApplication`, y esta llamada realiza la apertura del navegador con la `url` indicada como parámetro.

```
let url = URL(string: "https://www.thecircularlab.com")!
UIApplication.shared.open(url, options: [:])
```

2.4. Herramientas utilizadas

A continuación, se indicarán cuáles han sido las tecnologías y herramientas que se han utilizado para la realización del proyecto.

- **Librería Alamofire**

`Alamofire` es una librería escrita en `Swift`, creada para realizar peticiones `HTTP`. Proporciona métodos encadenables `request/response`, serialización `JSON` para el envío de parámetro en peticiones `POST`, validación de respuestas, y envío de peticiones `MultiparFormData` (como se ha visto en el apartado 2.3.3.3. Implementación Nueva Incidencia).

Para integrar esta librería en el proyecto se ha utilizado `CocoaPods`, que es un administrador de dependencias para proyectos `Cocoa`. Una vez instalado CocoaPods, se ha añadido la instrucción `pod 'Alamofire', '~> 4.5'` en el archivo `PodFile` del proyecto.

- **Xcode**

Se ha utilizado `Xcode` como `IDE` para el desarrollo de la aplicación, ya que es el `IDE` específico para el desarrollo de aplicaciones para `macOS`, `iOS`, `watchOS` y `tvOS`.

- **Swift**

Como lenguaje de implementación se ha utilizado `Swift` por imposición del cliente. Debido a que éste es un lenguaje de programación nuevo (2014), la mayoría de las librerías desarrolladas para utilizar en aplicaciones `iOS` están escritas en `Objetive-C`. Sin embargo, esto no ha sido un problema para el desarrollo de este proyecto, ya que las librerías que se han sido necesarias estaban escritas de forma nativa en `Swift`.

- **AVAudioRecorder**

La idea inicial para permitir al usuario grabar la descripción de la incidencia, fue crear un archivo de sonido. Para ello se usó la clase `AVAudioRecorder`, la cual permite grabar el audio del usuario y almacenarlo en el dispositivo. Tras múltiples intentos fallidos para enviar la grabación a través de una petición al servidor, y que en este se transformara ese archivo de audio en información textual, se decidió que la grabación de la descripción se almacenara directamente como una cadena de

caracteres. Para ello se ha usado la clase `SFSpeechRecognizer`, explicada en el apartado 2.3.3.3. Implementación Nueva Incidencia.

2.5. Pruebas

En la siguiente tabla se indican las pruebas realizadas y el resultado obtenido.

Prueba	Resultado
Inicio de sesión	Correcto
Registro	Correcto
Crear y enviar una incidencia	Correcto
Crear y guardar una incidencia	Correcto
Enviar una incidencia guardada	Correcto
Ver el listado de incidencias	Incorrecto

Ver listado de incidencias

El error se producía cuando el usuario creaba una incidencia y la guardaba. Esto hacía que la incidencia se almacenase en el dispositivo, y a su vez en el array de incidencias que representa a las incidencias del dispositivo. Cuando se entra en el detalle de una incidencia guardada, se puede enviar al servidor. Al enviarla, se eliminaba la incidencia del array de incidencias del dispositivo, pero no se actualizaba este cambio en el almacenamiento del móvil. Para ello, había que eliminar la incidencia del array, y luego sobrescribir el almacenamiento interno del dispositivo con el array modificado (sin la incidencia).

3. Seguimiento y control

3.1. Alcance y objetivos alcanzados

Con la realización de este proyecto se han alcanzado los objetivos propuesto en el análisis de requisitos de forma satisfactoria. Al finalizarlo, se ha obtenido una aplicación para dispositivos iOS que permite al usuario enviar incidencias a una base de datos, y un panel web de administración, para que los administradores gestionen las incidencias enviadas.

Al concluir el proyecto, se han creado los siguientes entregables:

- Aplicación para dispositivos iOS que permite al usuario enviar incidencias a un sistema de administración.
- Panel de administración web para poder gestionar las incidencias enviadas por los usuarios.
- Memoria en la que se describen las principales tareas del proyecto. Esta memoria contiene:
    - Memoria del proyecto
    - Anexo 1. Actas de las reuniones
    - Anexo 2. Planificación del proyecto
    - Anexo 3. Prototipos iniciales y diagrama de navegabilidad inicial
    - Anexo 4. Guía de estilos

3.2. Tiempo de ejecución real y desviaciones

En la tabla de la figura 18 se pueden ver las desviaciones en los tiempos de ejecución de las tareas, junto con el motivo de dichas desviaciones.

Durante el desarrollo del proyecto se han cumplido los plazos establecidos, excepto durante tres tareas: diseño de los prototipos iniciales; implementación del mapa para el panel de administración; y envío de incidencias desde la aplicación móvil.

El tiempo empleado para los prototipos iniciales ha sido mayor que el planificado ya que, en posteriores reuniones con el cliente, se solicitó añadir más interfaces a la aplicación (Inicio de sesión y Registro). Esto hizo que tuvieran que presentarse nuevos prototipos para su aprobación. Una vez aprobadas, se tuvieron que implementar esas nuevas interfaces, lo que a su vez cambió el diagrama de navegabilidad inicial.

La implementación del mapa para el panel de administración se vio retrasada debido a los problemas ocasionados por el `API` seleccionada para ello, ya que no está desarrollada para utilizarla con la versión de `composer` de nuestro proyecto.

Por ello, en lugar de crear el mapa a mostrar mediante un controlador, como sería la forma propia de `Symfony`, se implementó a través de `javascript`, directamente en la vista destinada a mostrar el mapa.

El envío de incidencias fue la parte más problemática del desarrollo de la aplicación móvil. En una primera implementación, se intentó realizar la petición `multipart` sin el uso de librerías externas, pero la conversión de la imagen a datos para ser enviada en la petición era errónea. Por ello, se optó por utilizar la librería **`Alomofire`**, la cual contaba con métodos específicos para realizar la subida de imágenes a servidor mediante peticiones `POST`.

Tarea	Horas previstas	Horas reales	Desviación	Motivo
Análisis	20	20	0%	Desviación nula
Diseño panel web	10	5	-50%	Para el diseño del panel se utilizó una plantilla
Diseño app iOS	20	25	25%	Cambios en los requisitos del cliente
Implementación panel web	90	95	5.5%	La implementación del mapa se intentó realizar con una librería
Implementación app iOS	120	125	5.2%	Investigación de las librerías para grabar el audio
Pruebas panel web	5	5	0%	Desviación nula
Pruebas app iOS	7	7	0%	Desviación nula
Memoria	30	40	33.3%	El tiempo planificado para la memoria no ha sido suficiente
Presentación	5	5	0%	Desviación nula
TOTAL	**307**	**327**	6.5%	No se planificó bien y no se dejó margen para imprevistos

Figura 18: Comparativa de horas planificadas y reales

En la figura 19 se puede ver gráficamente que las tareas a las que se dedicó un mayor número de horas fueron a la implementación de ambos proyectos. En el caso del panel de administración web, la implementación llevó más tiempo del esperado por los conflictos con las versiones de las librerías que se quisieron utilizar, y los intentos llevados a cabo para integrarlas en el proyecto.

En la aplicación iOS, el no tener experiencia hizo que en los primeros días se avanzara poco en el desarrollo. Las dificultades para grabar el audio del usuario y convertirlo a texto también supusieron un aumento en las horas de realización de esta tarea.

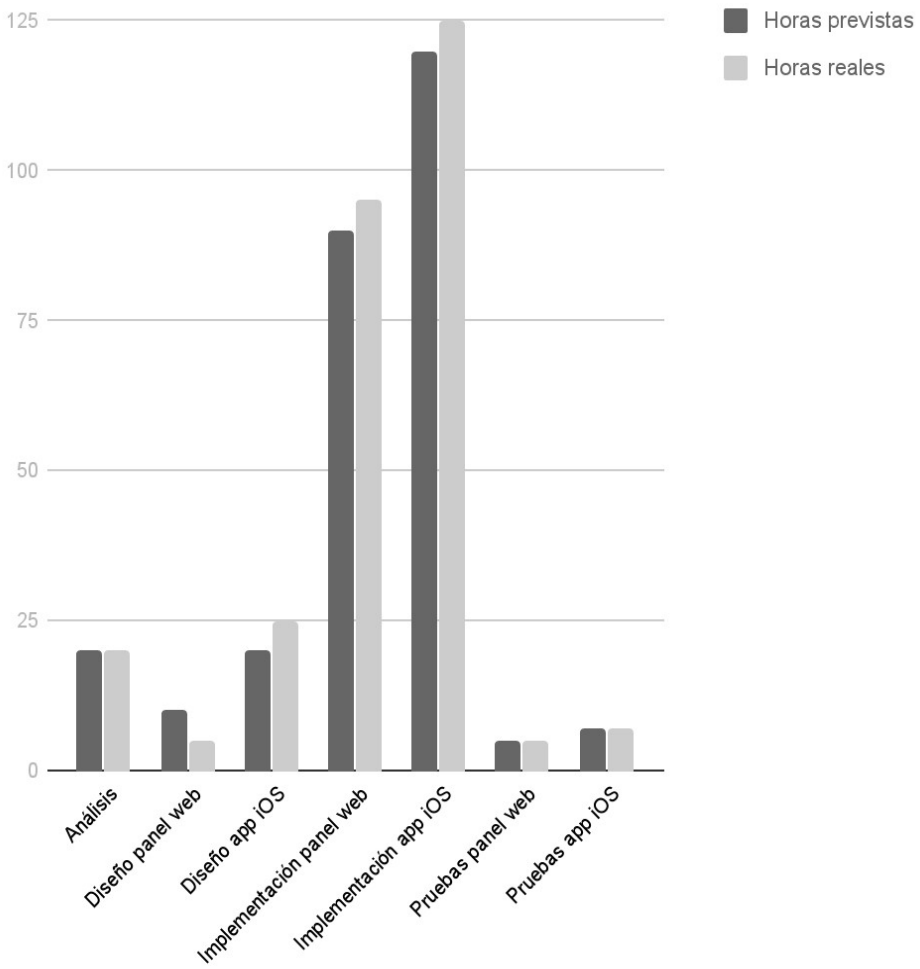

Figura 19: Comparativa horas previstas y reales

4. Conclusiones y mejoras

En este apartado se recogen las conclusiones recabadas a lo largo del proyecto, así como el aprendizaje personal y las posibles mejoras que se podrían implementar.

4.1. Conclusiones

El resultado general del proyecto ha sido satisfactorio, ya que se han logrado conseguir todos los requisitos establecidos por el cliente, sin una desviación importante en la fecha de entrega, a pesar de haber empleado un mayor tiempo de implementación del previsto.

Realizar este Trabajo de Fin de Grado ha sido un reto personal, ya que se trata de una aplicación para dispositivos `iOS`, y no contaba con experiencia desarrollando para este lenguaje. Cuando me propusieron este proyecto, pensé si debía aceptarlo o no, por desconocer la programación en `Swift`. Decidí aceptarlo ya que, aunque podría haber realizado el proyecto en `Android`, desarrollo con el cual tengo experiencia, creo que un Trabajo de Fin de Grado debe suponer un reto, y se debe aprovechar para demostrar que se puede crear una aplicación desde cero en un lenguaje de programación con el que no se tiene experiencia.

Los primeros días desarrollando con `Swift` me limité a seguir la guía de la página oficial de `Apple` para desarrolladores. Esto me sirvió para saber cuál era en ciclo de vida de una aplicación `iOS`, o los procedimientos por los que pasa una vista cuando va a ser mostrada al usuario. Un aspecto que supuso un punto de inflexión en el desarrollo de la aplicación fue darme cuenta de que para cada interfaz de la aplicación, se debe crear una clase que controle su comportamiento.

`Swift` no me resultó un lenguaje excesivamente difícil de comprender, en gran medida por el entorno de desarrollo `Xcode`. Al tener experiencia desarrollando en `Android`, no podía evitar compararlo con `Android Studio`, y `Xcode` facilita más el trabajo de cara al diseño y manejo de los componentes de una interfaz. Cuando se está trabajando con una pantalla de la aplicación, se divide el entorno de desarrollo en dos zonas, una con la interfaz y otra con el código de la clase que controla esa vista, y en todo momento se ve qué acción o qué variable está vinculada al control seleccionado en la interfaz.

Por último, el cliente ha expresado su satisfacción con el producto desarrollado en este proyecto. Si bien no es un resultado que puede ser explotado inmediatamente, el cliente ha mostrado su intención de utilizar este proyecto como base para el futuro desarrollo de una aplicación que comprenda todas las funcionalidades que desean, y de este modo crear una aplicación totalmente completa.

4.2. Aprendizaje personal

En este proyecto se han aprendido varios conocimientos técnicos, que se enumerarán a continuación.

1. Utilización del framework `Symfony` para la creación de aplicaciones web. Esta herramienta ha facilitado la organización de los distintos archivos de código, los recursos a usar por las vistas (archivos css y javascript), y proporciona un `ORM` compatible el gestor de base de datos que se ha utilizado.

2. La ventaja que proporciona usar un buen entorno de desarrollo. En este caso, si bien no ha habido posibilidad de elección, ya que `Xcode` es el único `IDE` que ofrece `Apple` para el desarrollo de aplicaciones `iOS`, no he sentido que necesitara buscar otro. Todas sus

herramientas están pensadas para que su manejo se lo más intuitivo y rápido posible para el usuario.

3. Utilización de `Composer`. Esta herramienta es el manejador de dependencias que utiliza `Symfony`, y es capaz de instalar las librerías que necesita nuestro proyecto con las versiones que necesitemos a partir de un comando y un archivo de propiedades. El uso de esta herramienta incrementa la usabilidad y portabilidad de la aplicación.

4. Añadir librerías al proyecto `Swift` mediante `CocoaPods`. `CocoaPods` en un administrador de librerías externas para proyectos `Objective-C` y `Swift` implementado en `Ruby`. Su instalación es muy sencilla (sólo hay que ejecutar el comando `sudo gem install cocoapods`). Esto creará un archivo `podfile` en nuestro proyecto. En este archivo se indican las librerías que se quieren integrar y se guarda. Una vez guardado, se ejecuta `pod install`, y las librerías quedan añadidas a nuestro proyecto.

4.3. Mejoras

A continuación, se exponen las posibles mejoras a realizar en el proyecto para incrementar el valor de la aplicación:

1. Permitir al usuario consultar la información de contacto de los distintos organismos encargados de gestionar las incidencias.

2. Que el usuario pueda hacer sugerencias acerca de nuevas categorías en las que clasificar las distintas incidencias que encuentren.

3. Cuando el usuario genere una incidencia, además de poder indicar las características de ésta mediante un formulario, lo podrá hacer a través de una grabación en la que proporcione los mismos detalles. Este segundo método será más ágil y rápido, ya que el usuario solo tendrá que proporcionar una fotografía descriptiva de la incidencia y una grabación con todos los datos necesarios (especificados en el requisito funcional 3 para iOS).

4. Una vez que el audio haya sido convertido en texto (cadena de caracteres) se utilizará NSLinguisticTagger, una clase usada en aplicaciones desarrolladas con Swift para dividir el texto proporcionado en lenguaje natural en párrafos, frases o palabras, y clasificar la información obtenida.

5. Bibliografía

- **PHP**

http://php.net/manual/es/intro-whatis.php

- **Symfony**

https://symfony.com/doc/current/index.html
http://symfony.es/libro

- **Swift**

https://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/
https://developer.apple.com/documentation/
https://itunes.apple.com/us/book/app-development-with-swift/id1219117996?mt=11

- **Stack Overflow**

https://stackoverflow.com/

- **Alamofire**

https://github.com/Alamofire/Alamofire
https://www.efectoapple.com/tutorial-alamofire-intro-1/

- **SFSpeechRecognizer**

https://medium.com/ios-os-x-development/speech-recognition-with-swift-in-ios-10-50d5f4e59c48

- **AVAudioRecorder**

https://developer.apple.com/documentation/avfoundation/avaudiorecorder

- **Egeloen/google-map**

https://packagist.org/packages/egeloen/google-map

- **Kingfisher**

https://github.com/onevcat/Kingfisher